Automata-Based Verification of Temporal Properties on Running Programs

Dimitra Giannakopoulou (RIACS) and Klaus Havelund (Kestrel Technologies)

Automated Software Engineering Group
NASA Ames Research Center, CA, USA
{dimitra, havelund}@email.arc.nasa.gov

Abstract

This paper presents an approach to checking a running program against Linear Temporal Logic (LTL) specifications. LTL is a widely used logic for expressing properties of programs viewed as sets of executions. Our approach consists of translating LTL formulae to finite-state automata, which are used as observers of the program behavior. The translation algorithm we propose modifies standard LTL to Büchi automata conversion techniques to generate automata that check finite program traces. The algorithm has been implemented in a tool, which has been integrated with the generic JPaX framework for runtime analysis of Java programs.

1 Introduction

Computer program correctness has, for decades, concerned industry and been studied in academia. Formal verification techniques such as theorem proving and model checking have been developed, that attempt to mechanize proofs that a program satisfies specifications expressed in some formal logic. However, such formal methods still remain to reach a state where they can be used in practice without considerable manual effort. A recent research direction is for model checking to be applied directly to programs written in standard programming languages such as Java and C [1, 2].

Although we find this work of great interest, we believe that a *light-weight* use of formal techniques will be useful and more practical in the shorter term. By "light-weight", we mean a method that is completely automatic, irrespective of the size of the examined program. Hence, the main concern is scalability: the technique should be practically applicable to large systems consisting of hundreds of thousands of lines of code.

An example of such a light-weight technique is what is often referred to as program monitoring. Here, the idea is to monitor the execution of a program against a formal specification written in some logic. This kind of technique is practically feasible since only one trace is examined, and it is useful since the logic allows stating more complex properties than is normally possible in standard testing environments.

In this paper, we describe an effort to develop such a technique for monitoring program executions against high-level requirement specifications written in Linear Temporal Logic (LTL). LTL is used to express properties in several model-checking environments [3]. Model checking an LTL property consists of generating a Büchi automaton for the negation of the property, and subsequently detecting cycles in the synchronous product of the model/program to be checked with the automaton.

Büchi automata are finite automata on infinite words. So the question naturally arises whether Büchi automata can be used to efficiently monitor finite traces of executing programs (since, either the program will terminate, or it will be interrupted at some stage). A typical way to model check a finite trace with Büchi automata is to extend the trace by repeating the last state indefinitely. However, we have found that the expressiveness of Büchi automata is not really required for checking finite traces.

The work presented in this paper provides a more efficient alternative to the use Büchi automata. We have developed an algorithm, based on standard LTL to Büchi automata construction techniques, which generates traditional finite-state automata that can be used to monitor efficiently LTL formulae on finite program traces. Our algorithm has been implemented in Trace analyZer (TaZ), an observer generator tool written in Java. TaZ has been integrated in Java PathExplorer (JPaX) – a generic tool for monitoring Java programs [4]. The result is an environment that can automatically check, on-the-fly, whether the current run of a Java program conforms to an LTL formula.

The remainder of this paper is organized as follows. Section 2 introduces LTL semantics on finite executions. It thus sets the theoretical grounds for Section 3, which presents our algorithm for generating LTL runtime observers. In Section 4, we discuss in more detail how TaZ generates observers based on our algorithm, and how the observers are used for program monitoring. Section 5 discusses related work, and Section 6 closes the paper with conclusions.

2 Finite-trace semantics for LTL

An executing program defines a sequence of states; an infinite execution can be viewed as an LTL interpretation, which assigns to each moment in time the set of propositions that are true at the particular program state. Model checking [5] detects infinite executions of finite-state systems through cycles in their state graphs; such executions define interpretations on which the truth-value of LTL properties is determined. Runtime verification, on the other hand, does not store the entire state-space of a program. Rather, it only observes finite program executions. For runtime verification, we therefore need to interpret LTL formulae on finite program traces.

We base our finite trace semantics on the following. Every LTL formula may contain a safety part or an eventuality part (or both). The safety/eventuality part requires that something bad-never/good-eventually happens in an execution. We modify the safety/eventuality requirement to mean that, *in the portion of the execution that we have observed*, nothing bad/something good happens. To reflect this, we modify the standard semantics of the temporal operators of LTL accordingly [6]. In what follows, we describe the syntax and finite-trace semantics of LTL as used in this paper.

The set of well-formed LTL formulae is constructed from a set of atomic propositions ($\wp$), the standard Boolean operators (not "!", and "$\wedge$", or "$\vee$", implies "$\rightarrow$"), and the temporal operator until "U". The following abbreviations are typically used: $\Diamond\varphi$ (eventually) for TRUE$U\varphi$, and $\Box\varphi$ (always) for $!\Diamond!\varphi$. Finally, we also use the temporal operator V which is defined as the dual of U, that is: $\varphi V\psi = !(!\varphi U\, !\psi)$.

Note that we are only interested in the next-free variant of LTL, namely LTL-X. This is typical in verification, because LTL-X is guaranteed to be insensitive to stuttering [5], which avoids the notion of an absolute next state. In the rest of this paper, the next-free variant of LTL is implied whenever we refer to LTL.

An interpretation of an LTL formula is a finite word $w = x_0 \ldots x_n$ over $2^{\wp}$(sets of propositions), where at some time point $0 \leq i \leq n$, a proposition p is true iff (if and only if) $p \in x_i$. We write w_i for the suffix of w starting at i. The semantics of LTL is defined as follows:

PROPOSITIONS: For $\varphi \in \wp$, $w \models \varphi$ iff $\varphi \in x_0$.
BOOLEAN OP: Standard semantics [7].
TEMPORAL OP: $w \models \varphi U\psi$ iff there exists $0 \leq i \leq n$ such that $w_i \models \psi$ and for all $0 \leq j < i$, $w_j \models \varphi$.

So the temporal operator U requires that ψ becomes true in the portion of the execution that has been observed, and that, of course, φ remains true in the interval that precedes the occurrence of ψ.

3 Algorithm

The goal is to construct a finite-state automaton that accepts exactly those finite words that satisfy a given LTL formula φ. Our algorithm is based on an efficient tableau-based LTL to Büchi automata translation presented in [7]. It generates an automaton for an LTL formula in two stages. First, it constructs a graph that represents the final automaton but without designated accepting states. Second, it selects accepting states. We remind the reader that the automata we generate are standard finite automata on finite words; as such, they accept finite traces that can lead them to an accepting state [8].

Construction: The first stage of our algorithm proceeds in essence in the same way as the corresponding stage of [7]. Given that, and due to space limitations, we will only briefly summarize this part of our construction. For more details, the reader is referred to [6, 7].

We deal with LTL formulae in negation normal form (all negations are pushed inside until they precede only propositional variables). The construction of the automaton is based on *expanding* a *graph node*. A graph node is a data structure that contains the following fields:

NAME: a unique name for the node.
INCOMING: the set of nodes that lead to this node, i.e., which have incoming edges to this node.
NEW: the set of LTL formulae that must hold on the current state but have not yet been processed.
OLD: the set of LTL formulae that have already been processed. Each formula in NEW that gets processed is transferred to OLD.
NEXT: the set of LTL formulae that must hold at all immediate successors of this node.

Field NEW in a graph node contains all the formulae that the node must make true. The idea of the expansion algorithm is to move formulae from NEW to OLD one by one by processing them in the following way. Each formula is broken down until we get to the literals (propositions or negated propositions) that must hold to make it true. For example, $\varphi \wedge \psi$ is broken down by adding both φ and ψ to the NEW field of the node. If there are alternative ways to make a formula true the node is split in two nodes, each of these nodes representing one way of making the formula true. For example, to make $\varphi \vee \psi$ true, we split the node into one node that makes φ true and another that makes ψ true. To satisfy temporal operator formulae, a node needs to also push obligations to its immediate successors. These obligations are stored in field NEXT, and are based on the following identities: $\varphi U\psi \equiv \psi \vee (\varphi \wedge X(\varphi U\psi))$, and $\varphi V\psi \equiv \psi \wedge (\varphi \vee X(\varphi V\psi))$.

When a node has been fully processed (i.e., the NEW field has become empty), the node represents a state of the automaton. Before being added to the automaton, it gets compared with all previously computed nodes/states. If the automaton already contains an equivalent node, the two nodes get merged. The notion of equivalence in our context is different from the one in [7].

Accepting states: As mentioned in [7], any path within the graph generated as discussed above is guaranteed, by construction, to satisfy all the safety conditions of the formula to which it corresponds. However, accepting conditions need to be imposed to make sure that eventualities are also satisfied. More precisely, we need to make sure that whenever some node contains $\varphi U \psi$, some successor node will contain ψ.

In our finite-trace semantics, we must ensure that any finite execution of the automaton that concludes in an accepting state (called an *accepting* execution) satisfies all the required eventualities. The eventualities that remain to be satisfied after any finite execution of the automata we generate, are reflected by the formulae contained in the NEXT field of the last state of this execution. This means that, unless there exist *U* formulae in the next field of the state, this state has satisfied its potential eventuality requirements.

Therefore, our algorithm designates as accepting those states that do not contain *U* formulae in their NEXT fields. The initial state is non-accepting, which reflects the assumption that traces contain at least one (initial) state.

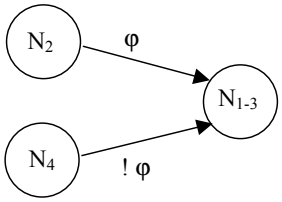

Fig. 1: Collapsing nodes with different OLD fields

Equivalent nodes: It can be seen from our construction that field OLD is used by our algorithm only to generate the labels of the automaton, but plays no role in the identification of accepting conditions. Therefore, two nodes are equivalent in our context when they have the same NEXT fields. Our algorithm typically generates fewer states than [7], since in [7] nodes can only be merged if both their NEXT and OLD fields are the same. This, however, does not make the OLD field redundant in our case; the literals contained in it are needed in order to determine labels of the automata edges. Therefore, our construction only stores literals (rather than any processed formula) in field OLD.

One needs to be careful during the process of collapsing equivalent nodes. If two nodes are collapsed, the resulting node must record the information of each component's INCOMING field and its associated OLD field. This is for it to record that it is obtained from alternative parent nodes by potentially different sets of literals. In the simple case where the literals in the OLD fields are the same, the OLD field of the resulting node is the same as the corresponding field of either of its components, and its INCOMING field is obtained as the union of their corresponding INCOMING fields.

For example, assume that a node N_1 with OLD={φ} (where φ is a proposition) and INCOMING={N_2}, is collapsed with node N_3 with OLD={!φ}, and INCOMING={N_4}. Let us call the resulting node $N_{1\text{-}3}$. This information is kept appropriately in node $N_{1\text{-}3}$, so that in the generated automaton, it will look as in Fig. 1.

Optimizations: Since the automata that our algorithm generates are finite automata on finite words, standard algorithms can be used to make these automata deterministic and minimal (details in [6]). For example, the deterministic and minimal automaton corresponding to formula $[](a \rightarrow \Diamond b)$ is illustrated in Fig. 2.

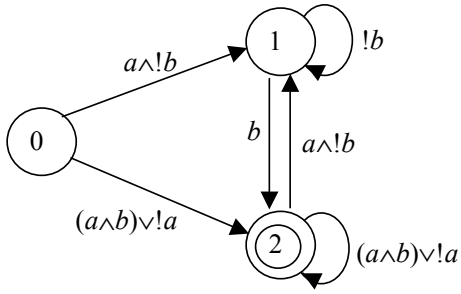

Fig. 2: Automaton corresponding to [](*a*→<>*b*)

4 Program monitoring

We have developed a tool, the trace analyser (TaZ), which receives as input an LTL formula, and generates an observer for traces of running programs, using the algorithm presented. TaZ has been integrated with the internally developed runtime-monitoring tool Java PathExplorer (JPaX) [4].

JPaX: JPaX is a general environment for monitoring the execution of Java programs. It consists of an instrumentation module, and an observer module.

Instrumentation is based on a script given by the user, which specifies the program variables to be monitored. The automated instrumentation inserts event-transmitting code after all updates to these variables. The updates are collected in an image state separate from the executing program state. The instrumentation script also defines a collection of Boolean valued proposition variables and an association between these and predicates over the observed program variables. When an image state-change

occurs, the propositional variables are re-evaluated, and what is sent to the observer is changes in these propositional variables. The Java byte code instrumentation is performed using the powerful Jtrek Java byte code-engineering tool from Compaq [9]. Jtrek makes it possible to easily read Java class files (byte code files), and traverse them as abstract syntax trees while examining their contents, and inserting new code.

The instrumented program, when run, emits relevant events to the observer. The observer may run on a different computer, in which case the events are transmitted over a socket.

TaZ observers: An observer in TaZ is a data structure that consists of the following fields:

- The automaton for the formula to be checked.
- The current states of the automaton. These may be multiple if the automaton is non-deterministic. Initially, the automaton is in its initial state.
- A hash-table that records the values, at the program state that is being verified, of the propositions involved in the formula.

The observer class implements the following interface, required by JPaX:

```
interface LTL{
  void init(STATEINIT init);
  void next(STATECHANGE change);
  void end();}
```

Method *init* is called by JPaX to pass the observer the values of propositions at the initial program state. Then, each time the proposition values change, JPaX calls the *next* method of the observer to pass it information about the state change. This is provided as a list of propositions that have changed value since the previous state.

Every time *next* is called, the observer performs the following steps. It updates the values of propositions in its local hash-table of the program state. It then checks which transitions rooting at its current states are enabled. To do this, it checks if the state of the program is compatible with the literals labelling these transitions. If, for example, *!a* labels a transition *trans*, and *a* is false in the current program state, then *trans* is enabled. The current states of the automaton are then updated to be the set of states that are reached through enabled transitions. If this set is empty, it means that the automaton cannot make a step, which reflects the fact that the property is violated by the specific trace of the program. This information is reported, which concludes the observer's job.

When the program is stopped, and if the observer is still running (i.e. it did not yet detect a violation or the fact that the property is satisfied), the program calls the *end* method of the observer. At this stage, the observer checks its set of current states. If there exists/does not exist at least one accepting state within this set, then the observer reports the fact that the property is satisfied/violated by the specific program trace, respectively.

Stuttering: As mentioned, the LTL-X variant of LTL is insensitive to stuttering. Therefore, the observer only needs to be notified whenever propositions in its alphabet change value. This can be implemented by the observer initially informing JPaX about the particular state attributes it is interested in observing.

Experimental results: We have used TaZ to generate automata for large formulae (more than 20 operators, mostly *U, V* and \/s, which cause nodes to split), and it produces results instantaneously. Checking program traces with the observers we generate is linear in the program trace, thus also very efficient.

We have applied our tools to artificially generated traces for early testing purposes. For properties that require checking the entire trace before a result is produced (e.g. $\Diamond\Box\varphi$), it takes our approach less than 5 minutes on a Pentium 4, 1.3 GHz processor, to process a trace 100 Million state-changes long. Currently, our tools do not implement the algorithms for making the automata generated deterministic and minimal; we expect that when these algorithms are incorporated, performance will be further improved.

Finally, we are also considering various ways of minimizing the effort required to compute enabled transitions [6].

5 Related Work

Program monitoring against specifications expressed in various logics has been investigated by several researchers. In [10], for example, the authors describe an algorithm for generating test oracles from specifications written in GIL, a graphical interval logic. Similarly to our approach, the oracles are based on automata. The generation is performed in two phases. During the first phase, a hierarchical non-deterministic automaton is computed, which, during the second phase, is turned into a classical deterministic finite automaton. The authors do not mention the application of minimization techniques to the resulting automata. The automata that they generate are typically larger than the ones that our algorithm computes. The reason is that they do not attempt to collapse equivalent states during generation.

An approach based on rewriting logic is presented in [11]. The authors have implemented in Maude [12] (an efficient rewriting logic system), rules that describe how an LTL formula is transformed by a new state encountered in the program, and how to decide, when the end of a trace occurs, whether the specification was

satisfied or not. This approach is less efficient than automata-based approaches, because it computes the transformations on the LTL formula during the analysis. Automata, on the other hand, are generated prior to analysis, and encode how state changes are triggered by inputs from program traces. Maude is, however, a powerful prototyping tool; it allows to easily define various types of logics. The authors have used it to also support past-time logics, for example [4].

The Temporal Rover (TR) [13] is a commercial tool for program monitoring based on temporal logic specifications. TR allows users to specify future time temporal formulae as comments in programs, which are then translated into appropriate Java code before compilation.

6 Conclusions

We presented an approach to generate deterministic and minimal finite-state automata used to check running programs against LTL specifications. The core of the algorithm modifies standard LTL to Büchi automata construction techniques. These techniques have been polished for efficiency over years of research. It has therefore been important for us to use these as a foundation. Moreover, we have been able to exploit standard algorithms for determinization and minimization of the automata we generate.

This approach is clearly more efficient than using Büchi automata for the same purpose. A benefit of our approach is that it does not require the detection of cycles in the product of the automaton with the program trace. Rather, all that is needed in terms of storage is the current state of the program, and the current state of the automaton. There are, therefore, no scalability issues involved. Additionally, we are able to generate minimal deterministic automata. Büchi automata provide full expressiveness only when they are non-deterministic. Finding the optimal (or approximately optimal) sized automaton for an LTL formula is PSPACE-hard [14].

An issue that occurs is whether LTL is the most appropriate language for expressing properties of running programs. LTL is a logic that has been widely used for expressing properties of reactive systems. This is particularly so in the domain of model checking. We believe that runtime monitoring and model checking will form components of extended debugging environments. It is therefore crucial to allow users to specify properties that are supported by both approaches.

From our experiments, the generation of observers is very efficient. So is their behavior during runtime analysis; specifications are checked in time linear in the length of the program trace that is examined. The core of our future research will therefore concentrate on how to improve the interaction of the running program with the observer so as to allow maximal independence between the two, but minimal disruption to the running program.

7 References

[1] Visser, W., Havelund, K., Brat, G., and Park, S. "Model Checking Programs", in *Proc. of the 15th IEEE International Conference on Automated Software Engineering (ASE'2000)*. 11-15 September 2000, Grenoble, France. IEEE Computer Society, pp. 3-11. Y. Ledru, P. Alexander, and P. Flener, Eds.

[2] Holzmann, G.J. and Smith, M.H., *Software model checking - Extracting verification models from source code.* Formal Methods for Protocol Engineering and Distributed Systems, Kluwer Academic Publishers, October 1999: pp. 481-497.

[3] Holzmann, G.J., *The Model Checker SPIN.* IEEE Transactions on Software Engineering, Vol. **23**(5), May 1997: pp. 279-295.

[4] Havelund, K. and Rosu, G. "Monitoring Java Programs with Java PathExplorer", in *Proc. of the First Workshop on Runtime Verification (RV'01)*. 23 July 2001, Paris, France, Electronic Notes in Theoretical Computer Science 55(2).

[5] Clarke, E.M., Grumberg, O., and Peled, S.A., *Model Checking*: The MIT press, 1999.

[6] Giannakopoulou, D. and Havelund, K., "Automata-Based Verification of Temporal Properties on Running Programs", RIACS, Technical Report 01.21.

[7] Gerth, R., Peled, D., Vardi, M.Y., and Wolper, P. "Simple On-the-fly Automatic Verification of Linear Temporal Logic", in *Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95)*. June 1995, Warsaw, Poland, pp. 3-18.

[8] Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*: Addison-Wesley, 1979.

[9] Cohen,S., http://www.compaq.com/java/download/jtrek.

[10] O'Malley, T.O., Richardson, D.J., and Dillon, L.K. "Efficient Specification-Based Test Oracles", in *Proc. of the Second California Software Symposium (CSS'96)*. April 1996.

[11 Havelund, K. and Rosu, G., "Testing Linear Temporal Logic Formulae on Finite Execution Traces", RIACS Technical Report TR 01-08, May 2001.

[12] Clavel, M*., et al.* "The Maude system", in *Proc. of the 10th International Conference on Rewriting Techniques and Applications (RTA'99)*. July 1999, Trento, Italy. Springer-Verlag, Lecture Notes in Computer Science 1631, pp. 240-243.

[13] Drusinsky, D. "The Temporal Rover and the ATG Rover", in *Proc. of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. August/September 2000, Stanford, CA. Springer, Lecture Notes in Computer Science 1885, pp. 323-330. K. Havelund, J. Penix, and W. Visser, Eds.

[14] Etessami, K. and Holzmann, G. "Optimizing Buchi automata", in *Proc. of the 11th International Conference on Concurrency Theory (CONCUR'2000)*. August 2000, Pennsylvania, USA, LNCS (Lecture Notes in Computer Science) 1877, pp. 153-167.